TP : Le problème de chargement de palettes

31 octobre 2017

Nous disposons de palettes de cartons de savons de Marseille contrefaits que nous souhaitons expédier dans des conteneurs depuis un port de Chine vers le grand port maritime de Marseille. Le coût du transport dépend uniquement du nombre de conteneurs utilisés, donc l'appât du gain nous incite à mettre le plus grand nombre possible de palettes dans chaque conteneur, pour minimiser le nombre de conteneurs utilisés. Tous les conteneurs sont de forme rectangulaire et de dimensions identiques. Les palettes ont aussi une base rectangulaire et ne peuvent pas être superposées. Nous allons donner une heuristique basée sur la programmation dynamique pour résoudre ce problème. On appelle *heuristique* un algorithme pour un problème d'optimisation qui calcule une solution sans garantir qu'elle est optimale.

Ce devoir est plus long que les deux précédents. Afin de compenser, une grande partie du code vous est fournie. Vous devez donc passer un peu de temps à comprendre les classes qui vous sont fournis avant de résoudre le problème posé. Cela fait partie de la démarche normale d'un développeur professionnel de poursuivre un projet déjà entamé, ce devoir vous sera donc aussi bénéfique dans votre étude de la programmation.

Le devoir dure quatre séances. Il doit être rendu sur Ametice, au plus tard le 7^{e} jour suivant le dernier TP à 14h (dates précises à voir avec votre responsable de TP). Vous rendrez pour chaque binôme tous les fichiers sources (y compris les fichiers que nous vous fournissons), des captures d'écrans de vos solutions, et si nécessaire un rapport décrivant vos travaux et vos difficultés. Tous ces fichiers doivent être dans un unique fichier archive d'extension `zip`, `tar` ou `tar.gz`. Merci de ne pas utiliser de formats de compression exotiques ou fermés.

1 Présentation du problème

Le conteneur va être représenté par un rectangle de grande dimension, et les palettes par des rectangles de plus petites dimensions appelés *tuiles*. Toutes les dimensions seront entières et inférieures à 100 unités.

1.1 Les tuiles

Une tuile est donc un rectangle, défini par une largeur l et une hauteur h. Nous disposons d'un nombre illimité de tuiles identiques à placer dans le conteneur. Nous allons donc devoir préciser l'emplacement d'une tuile, ainsi une tuile sera aussi définie par la position de son coin inférieur gauche, une paire d'entier (x, y). Les tuiles ne peuvent être placées que dans deux positions :

— horizontalement, avec le coin supérieur droit en position $(x + l, y + h)$,

— verticalement, avec le coin supérieur droit en position $(x + h, y + l)$. Cela revient donc à inverser h et l, et c'est ainsi que nous les représenterons.

Il est interdit de placer des tuiles dans d'autres directions. Les tuiles doivent donc être placées parallèlements aux axes, et les sommets doiventa voir des coordonnées entières. La Figure 1 présente trois tuiles et leurs coordonnées.

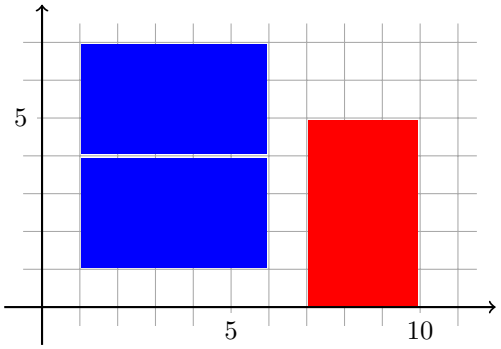

FIGURE 1 – 3 tuiles, de dimensions 5 par 3. Les tuiles bleues ($x = 1, y = 1, l = 5, h = 3$) et ($x = 1, y = 4, l = 5, h = 3$) sont placées horizontalement. La tuile rouge ($x = 7, y = 0, l = 3, h = 5$) est placée verticalement.

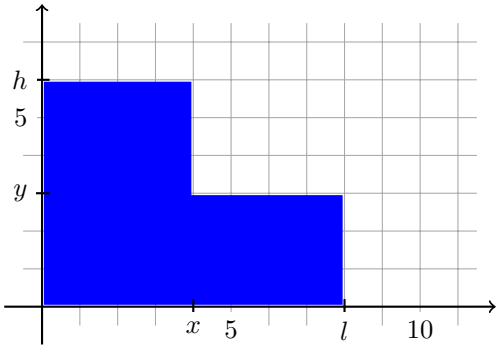

FIGURE 2 – Un exemple de L de coordonnées ($l = 8, h = 6, x = 4, y = 3$).

1.2 Le conteneur

Pour des raisons algorithmiques, nous allons considérer des conteneurs qui ont des formes plus générales que les rectangles. Les conteneurs auront un forme en L. Ils sont donc définis par la largeur l et la hauteur h d'un grand rectangle, plus la position de l'angle interne (x, y) du L. Le coin inférieur gauche est positionné en $(0, 0)$, et le conteneur est contenu dans le quadrant positif. Un exemple de L est donné en Figure 2.

Les rectangles sont des cas particuliers de L pour lesquels le coin interne est au niveau du coin supérieur droit.

1.3 Le problème

Étant donné un conteneur, défini par un L, et une taille de tuile (l, h), trouver un ensemble de tuiles de cardinalité maximum telles que :

— les tuiles sont disjointes les unes des autres (pas de superposition),

— les tuiles sont toutes contenues dans le conteneur.

Un ensemble ayant ces propriétés est appelé *paquetage* en mathématique (*packing* en anglais). La Figure 3 présente une solution pour un L défini par ($l = 15, h = 11, x = 9, y = 6$) et des tuiles de dimensions 7 par 2.

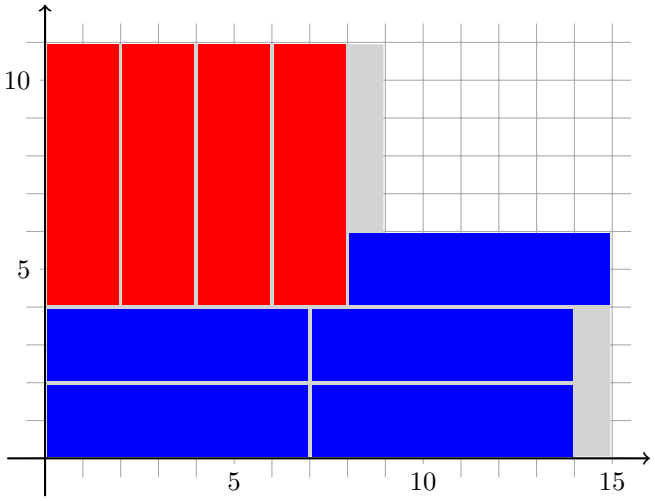

FIGURE 3 – Une solution au problème de conteneur sur une instance particulière. Ici le nombre de tuiles placées est 9.

2 L'algorithme

Nous utilisons un algorithme dynamique qui procède en coupant le conteneur en deux, puis en remplissant chacune des deux parties indépendamment par un appel récursif. Comment couper en deux ? L'algorithme teste toutes les façons de couper telles que les deux parties ont aussi la forme d'un L (ou d'un rectangle).

Il existe de multiples façons de couper un rectangle ou un L en deux rectangles ou Ls. La Figure 4 en présentent quelques unes. Pour une forme F de L donnée, notons $\mathcal{S}_F$ les différentes subdivisions de F en deux L plus petits. Les éléments de $\mathcal{S}_F$ sont donc des paires de L (F_1, F_2), tels que F_1 et F_2 peuvent s'emboiter pour redonner la forme F.

Nous pouvons alors décrire notre algorithme par une formule récursive :

$$\text{valeur}(F) = \max_{(F_1,F_2)\in\mathcal{S}_F} \text{valeur}(F_1) + \text{valeur}(F_2) \tag{1}$$

avec comme cas de base, si F est un rectangle de même dimension que la tuile, $\text{valeur}(F) = 1$. et si F a une aire plus petite qu'une tuile, $\text{valeur}(f) = 0$.

La complexité de l'algorithme dépend du nombre de sous-problèmes possibles, c'est-à-dire du nombre de formes de L, et du nombre de subdivisions de chaque forme. Un L est défini par 4 coordonnées, il y en a donc $O(n^4)$ en notant n la valeur maximum d'une dimension du L de départ. Il y a $O(n^2)$ rectangles. Le nombre de subdivisions est $O(n^2)$ pour les L, $O(n^3)$ pour les rectangles. La complexité pour évaluer cette formule, en utilisant la mémoïsation, est donc $O(n^4.n^2 + n^2.n^3) = O(n^6)$. Nous pourrons donc résoudre des instances avec $n \leq 20$ raisonnablement, difficilement au-delà.

2.1 Optimisations

Pour les plus grosses instances, il va falloir ajouter des optimisations. Principalement, nous allons essayer de ne pas avoir à tester toutes les subdivisions.

Pour cela, dans l'équation (1), après l'évaluation de chacune des subdivisions, nous allons tester si la solution trouvée est maximum. Si c'est le cas, nous n'avons pas besoin de tester les autres subdivisions et nous gagnons donc du temps. Sinon, nous continuons normalement avec les subdivisions restantes.

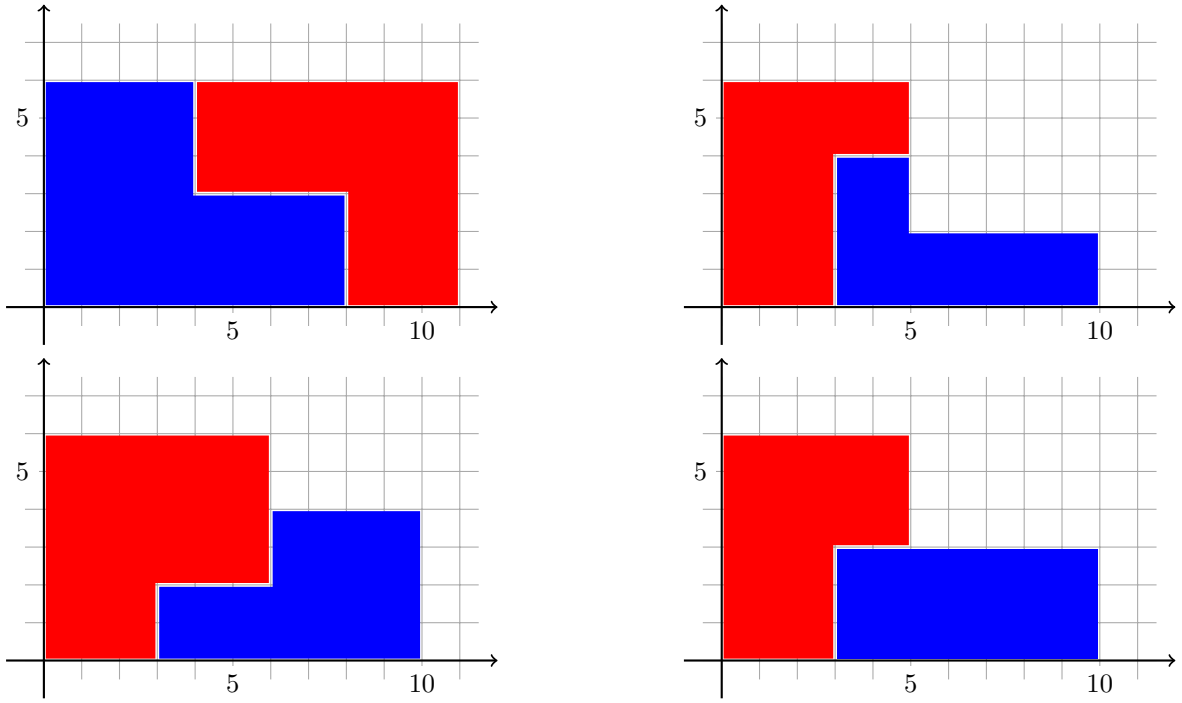

FIGURE 4 – Quelques subdivisions de quelques Ls.

Pour tester si une solution est optimale, nous utilisons une borne triviale sur le nombre de tuiles pouvant être placées : l'aire du L divisée par l'aire d'une tuile. Si une solution atteint cette valeur, nous savons que c'est une solution optimale.

Une autre optimisation est d'utiliser un algorithme glouton très rapide avant de tester les subdivisions. Si l'algorithme glouton retourne une solution atteignant la borne donnée par les aires, nous n'aurons pas besoin de tester la moindre subdivision.

L'algorithme glouton place toutes les tuiles horizontalement, ou bien toutes verticalement, et serrées le plus possible vers le coin inférieur gauche. La Figure 5 présente les solutions trouvées dans chaque cas sur un exemple. Si l'algorithme glouton ne trouve pas une solution atteignant la borne des aires, il faudra tester les subdivisions.

2.2 Plus d'information

Vous pouvez consulter la page web de chercheurs ayant optimisé encore plus cet algorithme et qui parviennent à résoudre des très grosses instances. Ils proposent aussi une applet en ligne pour calculer les solutions (ce qui permet de vérifier les votres).

`http://lagrange.ime.usp.br/~lobato/packing/`

3 Le code du projet

Nous vous fournissons une grande part du code nécessaire au projet à récupérer sur Ametice. Il comporte plusieurs classes que nous présentons maintenant. Certaines classes ne sont pas nécessaires au projet, mais utilisées avec JUnit pour réaliser des tests unitaires du code, elles vous permettront de vérifier que ce que vous écrivez est correct, ou du moins pas trop faux. Toutes les classes possèdent un

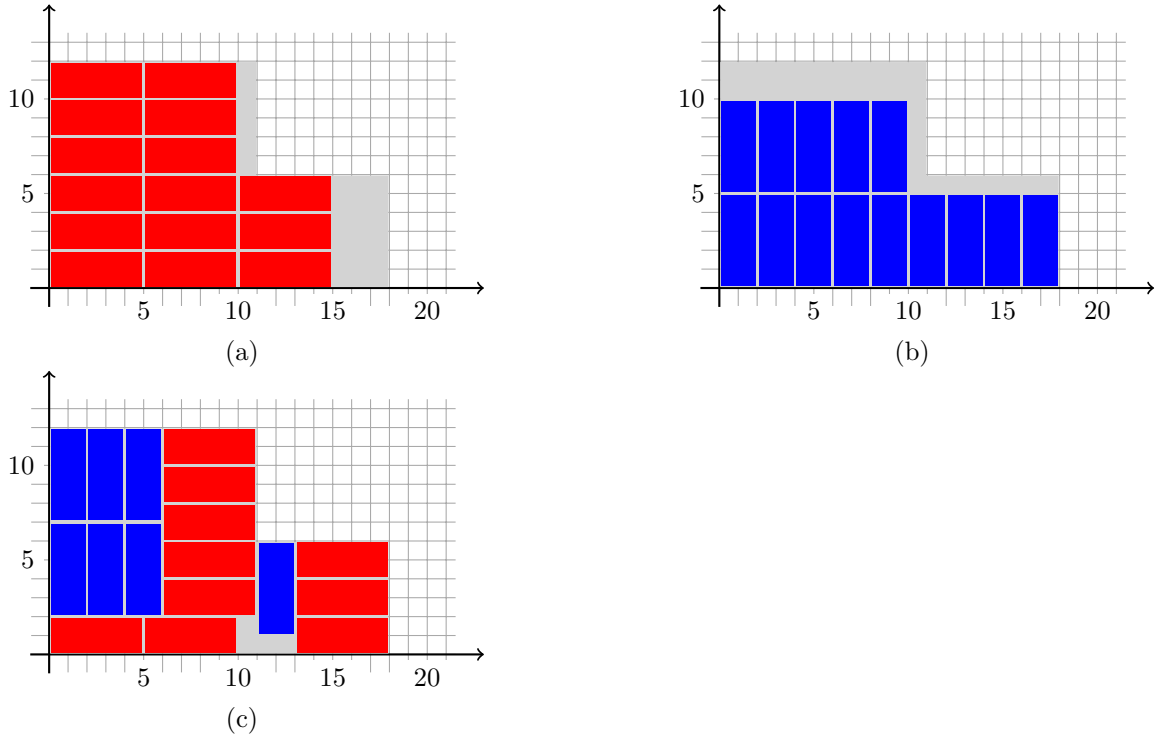

FIGURE 5 – En (a) une solution gloutonne horizontale de valeur 15. En (b) une solution gloutonne verticale de valeur 14. Les tuiles ont une taille de 5 par 2. Les solutions optimales ont une valeur de 17 et est représentée en (c) : l'aire non-utilisée fait 4 unités carrées, mais une seule tuile prend 10 unités carrées.

minimum de commentaires que nous vous invitons à consulter. Il n'est cependant pas nécessaire de lire tous les programmes, ne lisez que ce dont vous avez besoin.

Ce que vous avez à faire est résumé dans chaque classe par des commentaires `TODO`. S'il n'y en a pas, c'est qu'il n'est pas nécessaire de modifier la classe pour finir le projet. Seulement trois classes nécessitent d'être modifiées.

Le programme est découpé en deux packages :

geometry contient toutes les classes correspondant aux objets géométriques à manipuler : tuiles, formes en L, subdivisions, etc.

dynamicProg contient le solveur à base de programmation dynamique.

En plus de ces deux packages, la classe `DrawArea` gère le dessin et l'affichage des solution, et `Main` gère l'exécution du programme.

3.1 La classe `Tile`

La classe `Tile` encode les tuiles telles que définies dans le sujet. À noter qu'une tuile a une position dans le plan (ce qui n'est pas le cas d'une forme en L, qui est toujours positionné dans le quadrant positif avec un coin à l'origine).

En plus des accesseurs, une instance de Tile dispose des méthodes suivantes :

getArea calcule l'aire de la tuile.

flip permet d'échanger largeur et hauteur. Par contre le coin inférieur gauche ne bouge pas.

translate permet de déplacer la tuile selon un vecteur entier.

rotate effectue une rotation de la tuile avec pour centre l'origine du plan. Le paramètre indique le nombre de quarts de tour dans le sens trigonométrique (anti-horaire) de la rotation.

clone crée une copie de la tuile.

`Tile` est accompagnée d'une classe de tests unitaires `TileTest`.

3.2 La classe Packing

Puisque nous cherchons à placer autant de tuiles que possible dans le conteneur, il nous faut une structure de collections de tuiles pour représenter les solutions. C'est la classe `Packing` qui s'en charge, en utilisant une liste de tuiles. Elle propose les méthodes :

addTile pour insérer une nouvelle tuile.

transform pour appliquer une transformation géométrique (translation, rotation) à toutes les tuiles de la collection.

concat une méthode statique pour faire l'union de deux collections.

ainsi qu'un itérateur pour écrire des boucles for sur les éléments de la liste.

3.3 Les classes `EllShape`, `Subdivision`, `Part`

Une instance de `EllShape` permet de coder une forme en L. Elle possède alors les méthodes :

getArea calcule l'aire de cette forme.

getPackingUpperBound calcule pour une tuile une borne supérieure sur le nombre maximum de tuiles pouvant être disposée dans cette forme.

packGreedilyHorizontally place un nombre maximum de tuiles dans l'orientation horizontale.

packGreedilyVertically place un nombre maximum de tuiles dans l'orientation verticale.

packGreedily calcule pour une tuile de façon gloutonne une disposition de tuiles pour cette forme, en prenant le meilleur de greedyHorizontal et greedyVertical.

isRectangle détermine si la forme est un rectangle.

hashCode est une fonction de hachage pour les formes en L.

equals est un test d'égalité pour les formes en L.

En plus de ces méthodes, `EllShape` possède une autre méthode `subdivisions`. C'est la méthode la plus importante pour l'algorithme : elle calcule toutes les subdivisions possibles de la forme en deux formes plus petites. Une `Subdivision` est constituée de deux `Part`, chaque `Part` est une forme en L positionnée à un endroit du plan.

Les subdivisions sont données dans une `Stream<Subdivision>`. Les `Stream` sont, très approximativement, des listes de valeurs calculées à la volée. Dans le cadre de ce projet, on les utilisera simplement avec un itérateur :

```
Iterator<Subdivision> iter = ellShape.subdivisions().iterator();
while (iter.hasNext()) {
  Subdivision sub = it.next();
  doSomethingUsing(sub);
}
```

Ces classes sont testées à l'aide des deux classes `EllShapeAndPartTest` et `SubdivisionTest`, cette dernière testant spécifiquement l'algorithme de subdivision.

3.4 Les classes `DynamicProgrammingSolver` et `BestSolution`

La classe `DynamicProgrammingSolver` contient l'algorithme de programmation dynamique. La mémoïsation des solutions des sous-problèmes est assuré par une table de hachage, nous utilisons pour cela les `HashMap` de Java.

Le constructeur doit prendre la forme du conteneur et la tuile utilisée. Il initialise la table de hachage.

La méthode `computeOptimalSolution` calcule la solution optimale pour une forme donnée en argument. C'est la fonction récursive, elle fait appel à `retrieveOptimalSolution` pour trouver l'optimum sur les sous-instances.

La méthode `solve` résout le problème pour la forme de conteneur reçue par le constructeur.

Cette classe utilisée la classe `BestSolution` pour faire le calcul du maximum (dans la formule récursive). Elle garde donc la meilleure solution connue, et la met à jour dès qu'on lui donne une autre subdivision. Elle possède donc une méthode `accept` par laquelle on lui fournit toutes les subdivisions.

3.5 Les classes `Main` et `DrawArea`

Ces deux classes permettent la lecture du fichier d'instance, l'appel de la fonction de résolution, et l'affichage graphique de la solution trouvée. Il est inutile de les modifier. Le lancement de l'exécutable ouvre une fenêtre où sont affichées les solutions des instances décrites dans le fichier instances. Il suffit de cliquer pour passer à la solution de la prochaine instance (et d'attendre le temps du calcul). Chaque ligne de ce fichier définit une instance du problème par quatre entiers, dans l'ordre :

— la largeur du conteneur,
— la hauteur du conteneur,
— la largeur d'une tuile,
— la hauteur d'une tuile.

Les instances portent uniquement sur des conteneurs rectangulaires, mais vous pouvez modifier le `Main` si vous voulez travailler sur des formes en L.

4 Travail à effectuer

4.1 Mise en place du projet

Récupérez les sources du projet sur Ametice, et mettez-les dans le répertoire d'un nouveau projet dans Eclipse, puis commencez par configurer Eclipse, en suivant les instructions ci-dessous. En cas de difficulté, faites appel à votre chargé de TP.

JUnit Ouvrez le menu `Project, Properties`, allez dans la catégorie `Java build path`, onglet `Libraries`, et cliquez sur le bouton `Add Library`. Sélectionnez `JUnit`, puis `JUnit 5`. Ceci va ajouter au projet la librairie JUnit qui permet de faire facilement des tests unitaires du code que vous écrivez.

Nous vous fournissons plusieurs fichiers de tests, contenant notamment des tests pour les fonctions que vous devrez écrire. Pour lancer les tests, dasn l'explorateur de fichiers d'Eclipse, cliquez sur l'icône du répertoire `src-test` et sélectionnez `run as JUnit test`. Eclipse ouvre alors une vue JUnit, vous donnant le nombre de tests effectués, réussis, échoués, et une barre colorée : verte si les tests ont tous réussis, rouge sinon. JUnit donne aussi le temps pris par chaque fonction de test, ce qui peut être intéressant pour vérifier que vos programmes ne prennent pas trop de temps. Pour l'instant, certains tests sont au rouge, c'est normal, ils passeront au vert au fur et à mesure de votre travail.

Vous pouvez vous inspirer des tests déjà écrits pour en ajouter d'autres. Les tests que nous vous fournissont devront tous passer dans la version finale de votre projet (ce n'est pas le cas avec la version initiale).

Paramètres du programme Ouvrez le fichier `Main.java`, puis le menu `Run, Run configurations...` puis sélectionnez (dans la liste à gauche) application Java, ajouter, et renseignez `Main` comme classe principale. Allez dans l'onglet `Arguments`, et ajoutez dans la zone de texte `Program arguments` la ligne :

```
${project_loc}/resources/instances
```

Il s'agit du chemin désignant le fichier d'instances, la variable `project_loc` contient le chemin du répertoire contenant votre projet. Adaptez la ligne selon vos besoins. Cette opération permet donc d'ajouter un argument à la ligne de commande utilisée pour lancer l'exécutable, ici le nom du fichier d'instances.

Vous pouvez maintenant exécuter le Main, il devrait afficher des solutions très mauvaises pour les différentes instances. Vous pouvez aussi lancer les tests, et constatez l'échec de certains.

4.2 Implémentation

Commencez par compléter la classe `EllShape`. Les tests doivent tous réussir avant de faire la suite. Ensuite compléter les classes `BestSolution` et `DynamicProgrammingsolver`.

Vous n'aurez pas besoin de modifier les classes et méthodes qui ne sont pas marquées comme étant à modifier dans le code. Outrepassez cette règle à vos risques et périls.